# A syntax-independent code generation tool for IOPT-Petri nets

Fernando Pereira<sup>1,2,\*,†</sup>, Filipe Moutinho<sup>2,3,†</sup> and Luís Gomes<sup>2,3,†</sup>

<sup>1</sup>Instituto Superior de Engenharia de Lisboa, 1959-007 Lisbon, Portugal

<sup>2</sup>Center of Technology and Systems (UNINOVA-CTS) and Associated Lab of Intelligent Systems (LASI), 2829-516 Caparica, Portugal

<sup>3</sup>NOVA School of Science and Technology, NOVA University Lisbon, 2829-516 Lisbon, Portugal

#### Abstract

This paper presents a new code generation infrastructure for the IOPT-Tools framework, that automatically produces code in multiple languages from Input-Output Place-Transition (IOPT) Petri net models, to support the development of hardware/software controllers for embedded systems. The proposed infrastructure employs a two-step approach that starts with the creation of a language-independent XML document describing the execution semantics of the original Petri net model, which is later transformed into the syntax of the desired target programming or hardware description languages. The output of the first step is shared by all target languages, ensuring execution behavior consistency, independent of the chosen target language. As the IOPT simulation and model-checking tools also employ the code generated automatically, the proposed solution contributes to obtaining consistency between the behavior observed during simulation and model-checking and the final controllers running on embedded devices.

#### Keywords

Petri nets, automatic code generators, IOPT-nets, IOPT-Tools

#### 1. Introduction

The usage of design automation tools frameworks has a strong impact on the way the current system's development is performed, namely when considering the development of controllers for embedded and cyber-physical systems. Design automation frameworks have an important role in contributing to several societal challenges. Among those challenges, important to refer the contributions to digital transformation, as well as to reduce carbon emissions, contributing to performance, availability, and robustness improvements on final systems, as well as on wastes on materials used during the development processes. In this sense, design automation frameworks are completely in line with the goals of the "twin transition" concept, promoted within European initiatives combining digital transformations and decarbonising and sustainability approaches.

© 0 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Algorithms Theories for the Analysis of Event Data and Petri Nets for Twin Transition 2023, June 26–27, 2023, Lisbon, Portugal

<sup>\*</sup>Corresponding author.

<sup>&</sup>lt;sup>†</sup>These authors contributed equally.

<sup>☐</sup> fjp@deea.isel.ipl.pt (F. Pereira); fcm@fct.unl.pt (F. Moutinho); lugo@fct.unl.pt (L. Gomes)

D 0000-0001-7255-369X (F. Pereira); 0000-0002-0930-7418 (F. Moutinho); 0000-0003-4299-8270 (L. Gomes)

CEUR Workshop Proceedings (CEUR-WS.org)

The IOPT-Tools framework [1, 2], available online at http://gres.uninova.pt/IOPT-Tools/, has been successfully used over the years to develop controllers for embedded devices and to design digital systems. The systems are designed using IOPT-net [3] models, a class of non-autonomous Petri nets [4], that include input and output capabilities to communicate with the external world. In addition to the usual Places and Transitions found on traditional Petri net classes, IOPT models also have the concept of input and output signals and events. The input events are used to inhibit the firing of transitions and input signals are used to define guard conditions that also inhibit Transition firing. In the opposite direction, the marking of the Places may be used to activate output signals or assign new values to signals using output expressions, and transition firing may also trigger output events and modify the value of output signals. Signals may hold Boolean or integer values, to represent digital and analog values.

The IOPT-Tools framework contains a simulator tool used to test and debug models, presented in Figure 1, with a graphical user interface to manipulate the values of the input signals and events and observe the evolution of the system state and the resulting output signals. In addition, a state-space-based model-checking tool may be used to automatically verify certain model properties, including the reachability of desired states, the reachability of undesired states that could represent hazardous situations, or the existence of deadlocks.



Figure 1: The IOPT-Tools user interface (simulator page)

Using these tools, it is possible to debug and validate the models directly on the IOPT-tools user interface, and most errors are detected in the early design stages. This aspect is specially important, as any mistakes undetected before the prototype implementation may result in damaged hardware or cause hazardous situations.

After a model has been debugged using the software tools, the framework offers several automatic code generators that are employed to create software programs to run on embedded micro-controller boards, or hardware descriptions for Field Programmable Gate Arrays (FPGAs). In the prototype implementation phase, the inputs and output signals/events of the original IOPT model are associated with General Purpose Input/Output (GPIO) pins. This way, the controllers

are able to read sensors, command the status of mechanical actuators or read button/switch values, and control displays/LEDs to communicate with the users.

As mentioned before, any remaining undetected mistakes may cause expensive malfunctions that may damage the prototype hardware. This way, it is very important to avoid the introduction of new coding errors in the software/hardware that will run on the embedded devices. Compared to manual coding, the availability of automatic code generation tools was a major contribution to solve this problem. However, even the code produced by different code generation tools may exhibit small inconsistencies, that may lead to behavioral changes.

A common situation happens when a Petri net [4] model contains conflicts between Petri net Transitions that share common input Places: there are tokens to fire any of the Transitions, but not enough to fire all of them. In this situation, the code generated automatically must choose the Transitions that will fire and the ones that do not fire. The IOPT-net class offers Transitions priorities to help solve conflicts, always firing the transitions with the lowest priority value. However, if two transitions in conflict have the same priority, is the code generators that decide which transition fires.

Traditional autonomous Petri net classes typically solve this problem using a round-robin or random strategy. However, as IOPT-nets were created to support the design of embedded systems, the execution must be absolutely deterministic and must perform the same behavior consistently. This way, all the code generation tools must produce code with identical behavior, consistent with the simulation and model-checking tools.

This consistency is very difficult to ensure for code created manually, as each developer may sort the sequence of Transitions firing evaluation in different orders and even automatic tools may employ different ordering strategies. Although none of these strategies may be wrong, it is important to maintain consistent behavior between tools, to ensure that the resulting systems obey the same rules verified during simulation and model-checking. This problem is not exclusive to the Transitions evaluation sequence, as similar considerations could be applied to other aspects of the generated code, including the processing of input and output signals and events.

The automatic code generation infrastructure presented in this paper aims to contribute in this direction, by splitting the code generation into two steps. The first step is common to all languages and deals with the execution semantics of the models. It generates an XML document with a meta-program containing an algorithm that implements the model semantic rules. The second step deals with syntax and simply converts the language-independent XML to the syntax of the target language. This way, the code for the final implementation will share the first phase with the code used by the simulation and model-checking tools, ensuring the same behavior.

This paper has the following structure. The next section presents the IOPT-nets class, and section 3 presents related work. Then, in section 4, the proposed generation approach is described, followed by a generated code example in section 5. Finally, section 6 presents conclusions and future work.

#### 2. IOPT-nets

A formal description of the IOPT-net class can be found at [5]. Petri-net based formalisms offer many advantages to model discrete event systems, due to the capability to express parallelism and concurrency, in addition to a long list of mathematical properties. The IOPT-tools framework takes advantage of these properties to offer simulation, model-checking, and code generation tools.

The IOPT-nets inherit the characteristics of the low-level P/T nets, with the addition of input and output signals and events. It is a non-autonomous Petri net class, as the system state evolution is limited by external conditionalities: a) Transition firing can be inhibited by guard functions, consisting of logical expressions related to the values of input signals; b) Transition firing can also be inhibited with input events; c) Output actions associated with Places are used to define the value of output signals, that are only active when the places hold tokens; d) Transition firing can trigger output events, that may cause changes in output signals; e) Output actions associated with transitions may also change the value of output signals when the Transitions fire. The input and output signal can hold Boolean or integer range values, and are usually associated to physical GPIO pins on the hardware controller boards. Integer range signals are usually to represent analog values and Boolean signals are associated with digital GPIOs. This way, an IOPT-net model can read information from sensors, buttons, and switches and can also control physical actuators and motors, etc. As the main goal of IOPT-nets is the design of embedded system controllers, execution determinism is very important. This way, a maximal step execution semantics is used, meaning that all Transitions enabled and ready to fire, will fire on the next execution step. In order to help solve conflicts between transitions, the class also offers Test arcs, that prevent transition firing but do not remove tokens from the input Places and support Transition priorities.

Figure 2 presents a very simple example model, presenting 4 places (yellow circles) and 4 Transitions (cyan squares). This model also employs 3 input signals (cyan left arrows), two input events (cyan triangles), and two output signals (green arrows). The IE1 and IE2 input events are used to monitor changes in the IN1 and IN2 signals, detecting positive edges that happen when the signal changes from 0 to 1. These events are associated with Transitions T1 and T2 that can only fire when place P1 is marked and the respective event is triggered. The output signals OUT1 and OUT2 are associated with places P2, P3, and P4 by means of output actions. The value of OUT1 will be 1 when P2 is marked, 2 when P3 has tokens, and revert to a default value 0 when none of these Places is Marked. In the same way, OUT2 will have a value of 5 when P4 is marked or the default value is 0 otherwise.

Figure 3 presents the state-space (reach-ability graph) of the model presented in Figure 2, produced by the IOPT-tools model-checking tools. The model-checking tools detected 2 deadlock states (drawn as red) and two states containing Transition conflicts (drawn as magenta). This is a result of a design mistake in this model, whose correct execution depends on the sequence of external events: if events IN1 and IN2 occur in an alternate fashion, then both transitions T1 and T2 will fire. However, if two consecutive IN1 events occur (or two consecutive IN2 events) then one of the transitions T1 or T2 will fire twice, leading to a marking with two tokens on one of the P2/P3 Places and none in the other, that will prevent T3 from firing. This problem may be solved with two additional high-priority Transitions, to detect two tokens in



Figure 2: A simple IOPT-net model

one of the P2/P3 Places and inform the user about a possible error on the external hardware that generates EV1/EV2.



Figure 3: The state-space of the example model

### 3. Related work

The IOPT-tools framework has offered several automatic code generator tools for many years [1, 2], including support for the languages C, Javascript [6], VHDL [7], Matlab [8], and Instruction-

List [9] to generate code for Programmable Logic Controllers. Although the previous code generators offered by IOPT-tools were already based on XSLT transformations, each of these tools has an independent code-base and all generators start from plain PNML files. All of these tools were subject to extensive testing and efforts were made to check behavioral consistency. However, in case the same approach was used to create new code generators for other languages, it would require similar efforts for each language. In contrast, with the new approach, this work is greatly reduced as all generators share the same execution-semantics step.

Other tools employing Petri nets to design embedded system controllers, or to execute software code from Petri net models should be mentioned. For example, the CPN tools [10] framework allows the execution of Standard-ML code from high-level Colored Petri nets, and the Renew framework [11] combines Reference Petri nets with Java code. Other tools include CPN-AMI [12], Signal/Event nets [13], and Signal Interpreted Petri nets [14]. Finally, many works have been published that use code generated automatically using Petri nets. Some of these works include[15, 16, 17].

## 4. Automatic code generation

The execution of an IOPT-net model is performed in discrete steps that happen at a predetermined frequency. Each step starts by reading all input signals from the respective hardware GPIO pins, followed by the computation of input events according to changes in the input signals. Next, the Transitions are evaluated according to a predefined sequence, evaluating both the pre-conditions related to the marking of input Places and also the guard conditions and input events. When a transition fires, tokens are removed from the input places, and new tokens are added to output Places. After all, transitions have been evaluated, the new values of output signals and events are calculated according to the resulting marking and the transitions that fired. The evaluation sequence takes into account the Transition priorities and when multiple transitions have the same priority, they are sorted according to the respective name and identifier. This way, a deterministic sequence o evaluation is always respected.

As mentioned previously, the automatic code generation process is divided into two steps:

- 1. Execution semantics: In the first step, an XSL transformation examines a PNML file containing an IOPT model and produces an XML file containing a meta-program that implements the model execution rules. This step is shared by all code generators for the different languages: C, Javascript, VHDL, Python, etc.
- 2. Target language syntax: The second step uses other XSL transformations to convert the XML produces in the first step to the syntax of the desired target language.

Figure 4 illustrates the code generation process. In the first step, an XSL transformation receives a PNML file with an IOPT model and creates an XML file with a syntax independent description of a program to execute the model semantics. In the second step, different XSL transformations translate the XML document to the desired target language. The resulting code may be deployed on physical embedded devices, or may be used by the simulation (Javascript) and state-space generation (C code) tools. At the moment, these tools employ the code produced by the previous version of the code generation tools, which should be replaced by the new version.



Figure 4: The code generation process

The XML documents produced in the first step contain all the information required to build executable programs that implement the original model's behavior. It is divided into two sections: a header containing a description of the variables and data structures, and a code section with a set of procedures that perform several tasks. The data structures include variables to represent all input and output signals and events, data structures to hold the Petri net marking, and another data-structure to record the transitions that have fired. The execution algorithm employs three copies of the marking data-structure, one to memorize the previous marking, another to hold the remaining tokens after each transition fires, and another to store the new tokens created. At the end of each step, the remaining tokens are added to the newly created. The code section contains procedures to initialize all variables, initialize GPIO pins, read input signal values from GPIO, delay/pause between steps, compute input events, compute output signals associated with transition firing and marked places, and compute signals affected by output events. Another procedure performs a single execution step, evaluating all transitions and performing the corresponding token manipulation. Finally, a main procedure coordinates all the other procedures running a continuous loop. Instead of defining a touring-complete language, the XML documents produced in the first step employ just a minimal set of XML tags (presented in Table 1), required to execute IOPT models. Mathematical expressions, besides containing a list of operands and operators, may also contain multiple levels of sub-expressions.

In addition to the tags listed in Table 1, each tag has a set of configuration attributes. For example, variables have data-types, default values, and a min/max range. Operators have different types and operands may contain references to other variables or structure fields. Based on these tags it is possible to create different XSL transformations that convert the XML meta-

Table 1 XML tags

XML Tags	Meaning
<execution-semantics></execution-semantics>	Top node XML tag
<header></header>	Header defining variables and data structures
<code></code>	Code section containing procedures
<variable></variable>	Declared variables associated with signals and events
<struct> <field></field></struct>	Declare data structures and the respective fields
<data-table> <item> </item></data-table>	Declare tables containing data
<procedure></procedure>	Define procedures/functions containing a set of instructions
<call-procedure></call-procedure>	Call a procedure/function
<let> <variable field> <value> <expression></expression></value></variable field></let>	Assign a value to a variable or data-structure's field
<if> <condition> <then> <else></else></then></condition></if>	Conditional if-then-else
<loop></loop>	Define a loop containing instructions
<read-input></read-input>	Read an input signal from a GPIO
<read-event></read-event>	Read an input event from a GPIO
<write-output></write-output>	Write an output signal to a GPIO
<write-event></write-event>	Write an output signal to a GPIO
<expression> <operand> <operator></operator></operand></expression>	Mathematical expressions, operands, and operators

programs into the syntax of specific programming or hardware description languages. However, the second transformation step just performs a direct conversion from each XML tag to the corresponding code construct on the target language and does not apply any semantic changes. For example, a <struct> may be converted into a class or object in some languages, procedures may be translated to functions, and expressions are coded using the operators specific to each language. The only detail that may require attention, is the operator precedence rules that may differ from one language to another: in this case, the transformation may have to insert additional parenthesis in the resulting code.

#### 5. Generated code example

Applying the new code generator to the model in Figure 2, resulted in an XML executionsemantics document containing variables corresponding to each signal and event, data structures to store place-making and transitions fired, plus a series of procedures.

Some of the variables include:

```
<variable name="IN1" orig-node="signal" mode="input" type="boolean" value="0" io_pin="1" shift-register-depth="1"/>
<variable name="N2" orig-node="signal" mode="input" type="boolean" value="0" io_pin="2" shift-register-depth="1"/>
<variable name="N3" orig-node="signal" mode="input" type="boolean" value="0" io_pin="2" shift-register-depth="1"/>
<variable name="OUT1" orig-node="signal" mode="output" type="boolean" value="0" io_pin="4" wrap="0"/>
<variable name="OUT2" orig-node="signal" mode="output" type="boolean" value="0" io_pin="4" wrap="0"/>
<variable name="OUT2" orig-node="signal" mode="output" type="boolean" value="0" io_pin="5" wrap="0"/>
<variable name="L2" orig-node="signal" mode="output" type="boolean" value="0" io_pin="5" wrap="0"/>
<!-- Input and output events: -->
<variable name="L2" orig-node="event" mode="input" type="boolean"/>
<variable name="L2" orig-node="event" mode="input" type="boolean"/>
<!-- Net marking: -->
<struct name="marking">
```

Procedures to read and write inputs and outputs:

```
<procedure name="readInputs">
    <read -input variable="IN1" type="boolean" io_pin="1"/>
    <read -input variable="IN2" type="boolean" io_pin="2"/>
    <read -input variable="N3" type="boolean" io_pin="3"/>
    </procedure>
<procedure name="writeOutputs">
    </procedure name="writeOutputs">
    </procedure >
<procedure name="setup_io">
    </procedure = "Setup_io">
    </procedure name="pinMode" io_pin="1" mode="INPUT"/>
    <call -procedure name="pinMode" io_pin="2" mode="INPUT"/>
    <call -procedure name="pinMode" io_pin="3" mode="INPUT"/>
    <call -procedure name="pinMode" io_pin="5" mode="INPUT"/>
    <call -procedure name="pinMode" io_pin="5" mode="INPUT"/>
```

The procedure that executes a single step of the model contains code to evaluate the firing of each transition and the corresponding token consumption and creation. For example, transition T3 can only fire when both P2 and P3 contain tokens and the guard condition «IN3 = 1» is true:

```
< if >
  < condition >
    <operand type ="more-or-equal"/>
<operand type ="literal" value ="1"/>
    <operator type="and"/>
<operand type="variable" idRef="IN3"/>
    <operator type="equal"/>
<operand type="literal" value="1"/>
  </condition >
  <then>
<let type="struct" struct="transition_fired" field="t_3">
      <expression >
         </expression
    </let >
<let struct="avail_marking" field="p_2"></let
      <expression >
         <operand type="struct" idRef="avail_marking" field="p_2"/>
         < operator type = "sub"/ >
< operand type = "literal" value = "1"/ >
       </expression>
    </let >
<let struct="avail_marking" field="p_3">
      <expression >
         < operand type = "struct" idRef = "avail_marking" field = "p_3"/>
         <operator type="sub"/>
<operator type="literal" value="1"/>
       </ expression
     </let >
    <let struct = "new_marking" field = "p_4" >
       <expression >
         <operand type="struct" idRef="new_marking" field="p_4"/>
         <operator type="add"/>
<operator type="add"/>
<operator type="literal" value="1"/>
       </expression >
    </let>
  </then>
</if>
```

This XML document is subsequently transformed into the target language syntax. For example, the above XML fragment is transformed into the equivalent Javascript code:

```
if (avail_marking.p_2 >= 1 && avail_marking.p_3 >= 1 && data.IN3 == 1) {
    transition_fired.t_3 = Boolean(1);
    avail_marking.p_2 = (avail_marking.p_2 - 1);
    avail_marking.p_3 = (avail_marking.p_3 - 1);
    new_marking.p_4 = (new_marking.p_4 + 1);
}
```

The equivalent code produced using VHDL syntax is:

```
If (avail_marking_p_2 >= 1 AND avail_marking_p_3 >= 1 AND s_IN3 = 1)
Then
    transition_fired_t_3 := 1;
    avail_marking_p_2 := avail_marking_p_2 - 1;
    avail_marking_p_3 := avail_marking_p_3 - 1;
    new_marking_p_4 := new_marking_p_4 + 1;
End If;
```

The entire code can be consulted on the development area of IOPT-Tools, currently available at http://89.38.135.221/ fjp/iopt-tools/, opening the model "test.pnml" and executing the XML-2Step code generator tool. Currently, there are syntax transformations to produce C, Javascript, Python, and VHDL code.

# 6. Conclusions

The new approach to IOPT code generation offers advantages in terms of consistency in the output produced for different languages, as the first step, which deals with execution semantics, is common to all generators. The second step just performs direct syntactic translation and should not produce any semantic changes. This is specially important, as the simulation and model-checking tools make use of the Javascript/C code generated automatically. This way, the new solution contributes to ensure consistency between the code used during testing and debugging and the final version running on real hardware. Finally, the two-step approach greatly contributes to simplifying the creation of new code generators for different languages, that will reuse the existing execution-semantics XML and only require the development of a different transformation to the syntax of the desired language. Using this strategy, most of the code required to create a new code generator is re-used, contributing to reduce development time. As the second-step transformation does not deal with model semantics details, the developer can just focus on the target language syntax, meaning that possible design choices regarding open semantic details were already taken when the first-step transformation was created. Future work includes the creation of more transformations to generate code for other languages and porting the current IOPT simulator and model-checking tools to start using the code produced by the new code generator.

# Acknowledgments

This research was funded (in part) by the Portuguese FCT program, Center of Technology and Systems (CTS) UIDB/00066/2020/UIDP/00066/2020.

## References

 F. Pereira, F. Moutinho, A. Costa, J.-P. Barros, R. Campos-Rebelo, L. Gomes, IOPT-Tools – From Executable Models to Automatic Code Generation for Embedded Controllers Development, in: L. Bernardinello, L. Petrucci (Eds.), PETRI NETS 2022 – 43rd International Conference on Application and Theory of Petri Nets and Concurrency, Springer International Publishing, Bergen, Norway, 2022, pp. 127–138. Doi: https: //doi.org/10.1007/978-3-031-06653-5\_7.

- [2] F. Pereira, F. Moutinho, L. Gomes, IOPT-tools Towards cloud design automation of digital controllers with Petri nets, in: 2014 International Conference on Mechatronics and Control (ICMC), 2014, pp. 2414–2419. doi:10.1109/ICMC.2014.7232002.
- [3] L. Gomes, F. Moutinho, P. Fernando, J. Ribeiro, A. Costa, J.-P. Barros, Extending inputoutput place-transition Petri nets for distributed controller systems development, in: 2014 International Conference on Mechatronics and Control (ICMC), 2014, pp. 1099–1104. doi:10.1109/ICMC.2014.7231723, doi: https://doi.org/10.1109/ICMC.2014.7231723.
- [4] W. Reisig, Petri nets: an Introduction, Springer-Verlag New York, Inc., 1985.
- [5] L. Gomes, J. P. Barros, A. Costa, R. Nunes, The Input-Output Place-Transition Petri Net Class and Associated Tools, in: 2007 5th IEEE International Conference on Industrial Informatics, volume 1, 2007, pp. 509–514. doi:10.1109/INDIN.2007.4384809.
- [6] JavaScript Tutorial, https://www.w3schools.com/js/, 2023. Accessed: 2023-04-05.
- [7] V. A. Pedroni, Circuit Design with VHDL, MIT Press, Cambridge, MA, USA, 2004.
- [8] MATLAB MathWorks, https://www.mathworks.com/products/matlab.html, 2023. Accessed: 2023-04-05.
- [9] IEC 61131-3:2013, Programmable controllers Part 3: Programming languages, International Standard, International Electrotechnical Commission, 2013.
- [10] K. Jensen, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, volume 1, Springer Berlin, Heidelberg, 1997.
- [11] F. D. M. C. L. Kummer, O.; Wienberg, Renew User Guide, University of Hamburg, Department for Informatics, Theoretical Foundations Group, 2009.
- [12] A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, Y. Thierry-Mieg, New features in CPN-AMI 3: focusing on the analysis of complex distributed systems, in: Sixth International Conference on Application of Concurrency to System Design (ACSD'06), 2006, pp. 273–275. doi:10.1109/ACSD.2006.15.
- [13] P. Starke, H.-M. Hanisch, Analysis of signal/event nets, in: 1997 IEEE 6th International Conference on Emerging Technologies and Factory Automation Proceedings, EFTA '97, 1997, pp. 253–257. doi:10.1109/ETFA.1997.616278.
- [14] M. Minas, G. Frey, Visual PLC-programming using signal interpreted Petri nets, in: Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301), volume 6, 2002, pp. 5019–5024 vol.6. doi:10.1109/ACC.2002.1025461.
- [15] T. Mohammadpour, M. Yadollahi, A. Haghighat, A system modeling and simulation tools with capability of code generation using color-timed Petri net, 2009. doi:10.1142/ 9789814289986\_0006.
- [16] H. Dezani, N. Marranghello, A. S. Pereira, A. C. R. da Silva, M. Węgrzyn, Automatic Code Generation for Microcontrollers from Place-Transition Petri Net Models, IFAC Proceedings Volumes 44 (2011) 7873–7878. URL: https://www.sciencedirect.com/science/ article/pii/S1474667016448731. doi:https://doi.org/10.3182/20110828-6-IT-1002.02934, 18th IFAC World Congress.
- [17] F. Pereira, L. Gomes, Combining data-flows and petri nets for cyber-physical systems specification, in: L. M. Camarinha-Matos, A. J. Falcão, N. Vafaei, S. Najdi (Eds.), Technological Innovation for Cyber-Physical Systems, Springer International Publishing, Cham, 2016, pp. 65–76.